Uniswap V4 Hooks Guide (I): RBAC Hook

Umbrella Research

Follow

--

Listen

Share

Welcome to our Uniswap V4

hooks Guide!

This guide, supported by a grant from the Uniswap Foundation, aims to help developers understand Uniswap V4

hooks. It provides a walkthrough of the development of three new hooks. The guide consists of three blog posts, each building upon the previous article. By the end, developers will have an understanding of Uniswap v4

features and implementation patterns.

The beginning of our series starts with Captain Hook and a role-based liquidity pool that allows him to protect access to the trading chamber of his ship, the Jolly Roger.

This article will show how developers can implement role-based features within the Uniswap V4

protocol. The following two articles will provide insights into two additional aspects of Uniswap V4

: dynamic fees and the development of a full liquidity incentives hook.

You can follow the solution of the exercises by taking a look at the guide repository

GitHub - umbrellaresearch/uni-v4-hooks-tutorial: Code and exercises from the Uniswap v4 hooks…

Code and exercises from the Uniswap v4 hooks tutorial - GitHub - umbrellaresearch/uni-v4-hooks-tutorial: Code and…

github.com

Additional resources and documentation of the protocolcan be found here.

Let’s dive in!

1. Exploring role-based action pools

This exercise will explore role-based action pools in Uniswap V4

. These pools use hooks to restrict certain actions to a predefined set of addresses. The content will be divided into three sections: challenge, background, and development. In the challenge section, we will outline the problem. Then, in the background section, we will delve into the mechanics of Uniswap V4

and design our solution.

Finally, we will share the code for the designed solution.

The challenge

In our first exercise we will join our hero, Captain Hook, on his new adventure. He has discovered a rare amulet that grants access to the chamber of commerce on his magnificent ship, the Jolly Roger.

The amulet enables its owners to contribute gold and silver to the treasure chest, ensuring benefits from the ship’s trading. Additionally, Captain Hook wants to restrict trading in the chamber exclusively to pirates, so a specific type of credential is necessary.

To automate the verification process, Captain Hook decides to leverage the magic of Uniswap V4

. Let's assist him in fulfilling his desires!

We will work on establishing a Uniswap V4

pool that verifies the ownership of these credentials before allowing trading and contributing to the trading chamber. Let's begin!

The design

First, let's design our solution. Based on our requirements, we will need to check two separate credentials:

- The Amulet credential

checks if providing liquidity is allowed. If the user doesn't have the credential, then it reverts.

- The Pirate credential

checks if the address is allowed to swap. If the user doesn't have the credential, then it reverts.

Uniswap V4

introduces the concept of hooks, which enables developers to create pools with custom logic at different stages of the action lifecycle using pre-defined callbacks. Specifically, pools can implement one or more of the following callbacks:

- before / after initialize
- before / after modifyPosition
- before / after swap
- before / after donate

Our solution must check whether a user can provide liquidity or swap, so it will make use of the beforeModifyPosition

and beforeSwap

callbacks.

In Uniswap V4, every single pool is managed by a single contract, called thePoolManager

. To communicate with this contract, V4 uses a series of callbacks: you need to "acquire a lock" by calling the lock

function in the PoolManager

and then implement the lockAcquired

callback.

Let's explore how this architecture works in practice below:

We will interact using three different components:

- The Pool Operator contract

: This is a Smart Contract that we will create. It refers to the smart contract that requests the lock

and implements the lockAcquired

callback, where it will proceed to perform arbitrary pool actions (in our case, modifyPosition

and swap

).

- The

PoolManager

contract: This is the Uniswap V4 core singleton smart contract, that implements the necessary logic to provide locks to any caller that requests one, and offers all the different pool interaction actions available.

- The Hook contract

: This is another Smart Contract that needs to be created. It is the smart contract that will implement the hook callbacks, specifically our beforeModifyPosition

and beforeSwap

in our specific challenge.

Let’s explore how these three components interact in our project, in the scenario where a given user wants to provide liquidity to the pool:

1. The pool operator contract will request a lock to the PoolManager

. In return, the PoolManager

will invoke the lockAcquired

callback.

1. Upon receiving this callback, the pool operator will call the modifyPosition

action of the Pool Manager.

1. The PoolManager

will check if it needs to call the beforeModifyPosition

callback and invoke it.

1. Within the beforeModifyLiquidity

callback, the hook contract will verify if the user who initiated the action owns a specific NFT. The transaction will be reverted if ownership cannot be verified.

1. If the user does own the aforementioned NFT, execution will flow as expected into the actual modifyPosition

action.

The following diagram represents this interaction:

Now that we have designed our solution, it is time to build it!

Developing our solution

Below, we will follow a sequential approach, starting with the development of the interactions from the Pool Operator

to the Pool Manager

, and finish by implementing the hook callbacks.

To simplify matters, we will share examples of the modifyPosition

checks, given their similarities. However, the rest of the solution (using the swap

action) can be seen in the project repository.

1) The pool operator

Our pool operator contract needs to perform two different type of actions:

- First, it must request a lock

to the PoolManager

. In this interaction, it will be called by the user, so that user will be the msg.sender

.

- Second, it needs to implement the lockAcquired callback and decide which actions to perform. In this interaction, the msg.sender

will be the PoolManager

.

1) Requesting the Lock

Let’s start with the first part of the problem: creating a function that requests the lock. To do this, we will need to encode certain information, which will be passed as an argument to the lock

function. Let's take a look at what that information should be:

a) User address:

The user who initiated the action on the pool operator and is attempting to perform an action on the pool. This information is necessary as msg.sender

will be the PoolManager

on the lockAcquired

callback, so we cannot rely on it anymore in order to determine who the original initiator was.

b) Arguments for the

modifyPosition

function call:

After acquiring the lock

, the Pool Operator

will call the modifyPosition

function. We need to send all the necessary parameters for this action when requesting the lock so that it can be recovered on the callback. The modifyPosition

function accepts three different arguments:

1. PoolKey poolKey

: It uniquely identifies a pool within the PoolManager

.

1. ModifyPositionParams params

: A struct that specifies how the position should be modified.

1. bytes hookData

: Arbitrary data that will be used within the hook callback. We will use these bytes to encode our user address (the initiator), so that it can eventually be available on the beforeModifyPosition

callback implementation.

c) Function selector

We are implementing different actions in our smart contract (swap

and modifyPosition

) and we need to handle them differently. We need a way to differentiate them when they reach the lockAcquired

callback, so we achieve this by using an encodeCall

with the function selector of the desired function, along with the other information mentioned above:

2) Performing the action

The Pool Manager

will return with the lockAcquired

callback, forwarding the same data that was sent when the lock was requested. Since we sent an encoded call, we can simply relay that call to the target function performModifyPosition

.

Finally, after acquiring the lock, we can callthe modifyPosition

function of the PoolManager

. When called, the Pool Manager

will in turn call the beforeModifyPosition

callback, which will be created in the next section.

3) Settling balances

In the architecture of Uniswap V4

, there is a critical component called the delta

field. This field tracks the balances owed to the pool (positive) or to the user (negative) during a lock. Before the lock

can be released, the accumulated deltas must be neutralized to zero. In our scenario, we only need to verify if the delta amounts for both tokens are different from zero, and adjust the balances accordingly.

The fact that we know deltas will be different from zero at the end of the lock allows the pool manager to utilize transient storage opcodes, as proposed in EIP-1153. This enables the Uniswap v4 architecture to achieve gas efficiency.

2) The hook

Now that we have our Pool Operator

contract in place, we are ready to start developing our hook contract.Uniswap's v4-periphery provides an abstract contract called BaseHook that can be extended to develop hooks on top of it. We will get started by extending that contract.

This BaseHook

contract includes the getHooksCalls

function, that needs to be overridden. It will inform the PoolManager

about the subset of hooks that this contract will implement.

Hooks.Calls

is a struct made of booleans, indicating which hooks need to be invoked during execution. We are going to check permissions before swapping and before modifying a liquidity position, so let's fill it out:

Now are ready to start working on the actual logic for our hook callbacks. The logic should perform the following tasks:

- Verify that the user does own the NFT amulet before providing liquidity using the beforeModifyPosition

callback

- Confirm pirate credential ownership before executing a swap using the beforeSwap

callback.

Let's focus on showcasing the beforeModifyPosition

callback. This function receives the following arguments:

- sender

: the address of the lock owner (the Pool Operator

)

- poolKey

and modifyPosition

parameters, which are not necessary for this use case

- hookData

parameter, including the data we added when the lock was created, containing the user's address

To handle this callback, we need to check two conditions:

1. Ensure that the address that created the lock is our Pool Operator,

to prevent hookData

manipulation

1. Ensure that the user is the owner of the amulet, by checking the NFT balance. We will use an ERC-1155 token for it, called items

We can apply the exact same strategy to check the permissions beforeSwap

, checking the balance of the pirate credential instead:

3) Testing the Hook

Finally, it's time to test the results! We will conduct a Foundry test to verify if an external user who owns the required NFT can modify their liquidity position. Let's proceed step by step, beginning with the setUp

:

First, we need to deploy our hook. Uniswap V4

determines which hooks to invoke by analyzing the leading bits of the address where the hooks contract is deployed. We will provide more detailed instructions on how to deploy those hooks in future blog posts. However, for the purpose of this tutorial, we have encapsulated that logic into a deployHook

function.

Now that we have deployed our hook, let's initialize the state for our users. We will have three different users: one who owns the amulet, another who is a pirate credential owner, and finally, an unauthorized user.

As a last step, we can run the tests to verify the contract worked as intended.

Et voila! We created our permissioned pool using hook callbacks!

Conclusion and next steps

Well done on completing the initial phase of our Uniswap V4

hooks journey! You have skilfully sailed through the challenging yet fulfilling realm of role-based action hooks, akin to Captain Hook's quest for treasure in uncharted waters. With the fundamental knowledge you have gained, you are now ready to dive into the more advanced facets of Uniswap V4

.

Remember, this is just the beginning. You have taken the crucial first step towards mastering the art of creating secure, efficient and powerful decentralized exchanges. The skills you have honed will be invaluable as you continue to expand your DeFi toolkit.

Looking ahead, our adventure will shift focus to the world of hook managed fees. And that's not all. We will also take you through the full process of shipping a project, where you will apply everything you have learned to build a comprehensive Liquidity Incentives Hook.

Stay tuned for the next chapter of our journey, where the adventures intensify and the challenges grow even more thrilling. Keep your coding tools at the ready and maintain your pirate spirit for the upcoming Uniswap V4 excitement!